Signed and Unsigned Types in Interfaces

Suppose you're writing a class template for array-like objects. For consistency with built-in arrays, it would be appropriate to define a constructor taking an argument specifying the size of the array. Assuming the number of elements in Array objects can change (through, for example, assignment or explicit size-changing member functions), it would also be convenient to provide a function that returns the current size of an Array object.

If you're like most C++ programmers, your first cut at declaring these member functions would look something like this:

```
template<class T>
class Array {
public:
  Array(int size);
  ...
  int length() const;
};
```

Upon further reflection, however, you might conclude that it makes no sense to have an array whose size is less than zero, so you might modify your class definition to specify more precisely that only non-negative numbers are expected:

```
template<class T>
class Array {
public:
  Array(unsigned int size);
      // size is now unsigned
  ...

  unsigned int length() const;
      // return type is now unsigned
};
```

This design makes your rejection of negative array sizes clear, and, as a bonus, it allows you to double the maximum size of your Array objects without having to resort to long ints. Furthermore, it follows the lead of C++ itself, which generally employs unsigned types when specifying how many of something are needed. For example, operator new uses the unsigned type size_t for specifying how many bytes of memory to allocate, and the standard vector classes [citations] use the typedef size_type as a synonym for whatever unsigned type is used to specify how many elements a vector should initially contain.

These features make unsigned types attractive, but there is a dark side to their use that must also be taken into account. One problem is that unsigned types tend to decrease your ability to detect common programming errors. Another is that they often increase the likelihood that clients of your classes will use the classes incorrectly.

Consider first error detection. Suppose a programmer defines an `Array` object as follows:

```
int f();       // f and g are functions that return
int g();       // ints; what they do is unimportant

Array<double> a(f()-g()); // array size is f()-g()
```

There's nothing wrong with this definition for `a`, except for the possibility that the programmer writing it made an error. Instead of the size of the array being `f()-g()`, perhaps it should have been `g()-f()`. Or maybe it should have been `f()+g()`. Or possibly it should have been `f()-g()+1`; off-by-one errors are certainly common enough. Heck, it's even possible that `f()-g()` is correct, but `f()` or `g()` is returning a bad value. Any of these scenarios could lead to the `Array` constructor being passed a size that was less than zero. As the author of the `Array` class, there's no way you can keep clients from making mistakes such as these, but you can do the next best thing: you can *detect* such errors and act on them.

Well, actually, maybe you can't. You can't if you declared `Array`'s constructor to take an unsigned value, because if a negative number is passed to a function as an unsigned, the number seen by the function isn't negative at all. Instead, it's a very large positive number. As a result, you'd have no way within the `Array` constructor of distinguishing between a large, but valid, positive size value and a large, but invalid, positive size value arising from passing in a negative number. Hence, you'd be unable to detect programming errors that result in negative array sizes being requested. Because such errors are not uncommon, this makes your class easy to use incorrectly. Well-designed classes are easy to use correctly and hard to use incorrectly, so you should reconsider your design with an eye toward allowing the `Array` constructor to detect when an `Array` with a negative size is requested.

The easiest way to do this is to revert to the original function prototype, the one where the `Array`'s size is passed as a signed `int`. Detection of negative sizes is then no challenge at all:

```
template<class T>
Array<T>::Array(int size)
{
  if (size < 0) {
    throw an exception or take some other action;
  }

  ...

}
```

The use of a signed parameter type in the `Array` constructor makes this kind of sanity checking possible. The conclusion should be clear: don't use unsigned types in interfaces unless you are willing to forego the ability to detect negative values passed through those interfaces.

Now, the design using signed parameter types only *allows* error detection, it doesn't *require* it. For example, if you don't want to pay for error detection in production code, you can use conditional compilation to eliminate the cost:

```
template<class T>
Array<T>::Array(int size)
{
#ifdef DEBUGGING
  if (size < 0) {
    throw an exception or take some other action;
  }
#endif

  ...

}
```

In many cases, such debugging code exacts a negligible performance penalty, so you can often allow it to remain in production code. That's especially important for class libraries that are used by many clients, because the errors being detected in this case are in *client* code, not library code. Such errors are almost always worth detecting, even in production versions of programs and libraries.

It's now clear that unsigned types lead to difficulties when used in parameter lists, but what's the problem with using them for the return type of functions like `Array::length`? `length` is a member function, so you, as class implementer, have complete control over how it computes its result. You also have control over whether that value is allowed to be negative. If you're careful, then, the value will never be negative, and there's no problem with returning an unsigned, right?

Not right. The problem again lies with those pesky clients, You just can't rely on them to do what they're supposed to do. In particular, you have to assume they'll write code like this:

```
Array<double> a(f()-g());     // as before, but
                              // assume f()-g() > 0
...

int arraySize = a.length();   // initialize an int
                              // with an unsigned
```

It's a fact of life that many programmers never venture beyond `int` in the world of integral types, so the likelihood is high that if `Array::length` returns an unsigned value, many clients will save that value as an `int` anyway. If they do, and if the value returned by the function is greater than the maximum value that can be stored in an `int`, they'll end up storing a negative number. Something thereafter will almost surely break, and though the fault will technically be theirs, you will have facilitated the error by returning a type that is prone to lead to errors. In many cases, it's better to simply return an `int` directly, thus nipping the problem in the bud.

If you do that, of course, you reduce the range of values you can return, because you must pay for a sign bit you don't really need. In many cases, however, the increase in software robustness more than compensates for the reduced range of values.

References

B. Stroustrup, "Making a Vector Fit for a Standard," *C++ Report*, October 1994.

Accredited Standards Committee X3J16, "Working Paper for Draft Proposed International Standard for Information Systems — Programming Language C++," April 1995.